John von Neumann Institute for Computing

A Java/Jini Framework Supporting Stream Parallel Computations

M. Danelutto, P. Dazzi

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 681-688, 2006.

http://www.fz-juelich.de/nic-series/volume33

A Java/Jini framework supporting stream parallel computations

M. Danelutto[a] & P. Dazzi [b c]

[a]Dept. Computer Science – University of Pisa – Italy

[b]ISTI/CNR – Pisa, Italy

[c]IMT – Lucca Institute for Advanced Studies – Italy

JJPF (the Java/Jini Parallel Framework) is a framework that can run stream parallel applications on several parallel-distributed architectures. JJPF is a distributed execution server, actually. It uses JINI to recruit the computational resources needed to compute parallel applications. Parallel applications can be run on JJPF provided they exploit parallelism accordingly to an arbitrary nesting of task farm and pipeline skeletons/patterns. JJPF achieves almost perfect, fully automatic load balancing in the execution of such kind of applications. It also transparently handles any number of node and network faults. Scalability and efficiency results are shown on workstation networks, both with a synthetic (embarrassingly parallel) image processing application and with a real (not embarrassingly parallel) page ranking application.

1. Introduction

It is generally assessed that real parallel applications usually exploit parallelism according to a limited, well-known set of patterns (or skeletons) [9,21,20,7]. With the advent of grids [13,14] and large cluster architectures [25] some of the parallelism exploitation patterns originally proposed in the skeleton framework have been extensively used to implement high performance, parallel grid applications. Indeed, very often parallel applications are programmed exploiting *by hand* typical grid middleware or operating system/distributed framework mechanisms without even stating they owe to the algorithmic skeleton or parallel design patterns most of the techniques use to exploit parallelism. An example of parallelism exploitation pattern that is very often used both in grids and in more traditional distributed frameworks is the task farm one. In a task farm, a set, or a stream, of independent tasks are computed to obtain a set of results. A single program or function is used to compute all the results out of the input tasks. Such parallelism exploitation pattern is also referred to as *embarrassingly parallel computations* [27]. All the parameter sweeping applications, that is those applications that "try" input data sets to find out the best one with respect to some measure function, can be easily programmed exploiting parallelism according to the task farm pattern. Also, most of the grid applications that can be programmed using tools such as Condor [11] (basically a batch job scheduler) can be programmed as task farm instances. Another well-known and used parallelism exploitation pattern is the pipeline one. In a pipeline a set of input tasks are processed by a set of stages. Each stage just computes a result out of the result provided by the previous stage and delivers result to the immediately following stage. Task farm and pipeline parallelism exploitation patterns are often referred to as stream parallel (or task parallel) patterns/skeletons [23,20]. We already demonstrated that arbitrary compositions of pipeline and task farm patterns can be efficiently implemented, with respect to service time, using their normal form, that is transforming the original skeleton tree/composition into a program that is basically a task farm with sequential workers [3].

[0]This work has been partially supported by Italian national project no. RBNE01KNFP *GRID.it* and No. 02.00640.ST97 and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

Overall, this allows us to conclude that if we succeed providing a distributed environment efficiently exploiting task parallel computations, this will be very useful to implement different applications in really different applicative and hardware contexts. Our group already published several works related to the implementation of such kind of programming environments [7,4,12,8] and it is currently involved in a large national project (the Italian FIRB project GRID.it [17]) aimed at designing and implementing a prototype, high performance, structured programming environment [26,2,1]. In this work, we discuss a parallel programming framework (JJPF) built on top of plain Java/Jini that can run stream parallel applications on several parallel/distributed architectures ranging from tightly coupled workstation clusters to generic workstation networks and grids. The framework directly inherits from Lithium and `muskel`, two skeleton based programming environments we previously developed at our Department [4,12]. Both Lithium and `muskel` exploit plain RMI Java technology to distribute computations across nodes, and rely on NFS (the network file system) to distribute user code to the remote processing elements. JJPF, instead, is fully implemented on top of Jini/Java and relies on either Jini/Jeri class loaders or on a brand new, hybrid class loader package, to distribute code across the remote processing nodes involved in stream parallel application computation. JJPF exploits the stream parallel structure of the application in such a way that several distinct goals can be achieved: a) *load balancing* across the computing elements participating in the computation b) *automatic discovering and recruiting* of processing elements available to participate to the computation of stream parallel applications exploiting standard Jini mechanisms c) *automatic substitution of faulty processing elements* by fresh ones (if any). Therefore the stream parallel applications computations resist to both node and network faults. The programmer does not need to add a single line of code in his application to deal with faulty nodes/network, nor he has to take any other kind of action to get advantage of this feature. JJPF has been tested using both synthetic and real applications, on both production workstation networks and on a blade cluster, with very nice and encouraging results, as described in Section 3.

2. JJPF

JJPF has been designed to provide programmers with a user-friendly environment supporting the efficient execution of stream parallel applications on a network of workstations, exploiting plain, state of the art, Java technology. Overall JJPF provides a distributed server providing a stream parallel application computation service. Programmers must write their applications in such a way they just exploit an arbitrary composition of task farm and pipeline patterns. Task farm only applications are directly executed by the distributed server, while applications exploiting composition of task farm and pipeline patterns are first processed, in a completely automatic way, to get their normal form [3] and then their normal form is submitted to the distributed server for execution. Using JJPF, programmers can express a parallel computation exploiting the task farm pattern simply using the following code:

```
BasicClient cm = new BasicClient(program,null,input,output); cm.compute();
```

provided that `input` (`output`) is `Collection`of input (output) tasks and `program` is an array hosting the worker code of the farm. The worker code is a `Class` object relative to the user worker code. Such code must implement a `ProcessoIf` interface. The interface requires the presence of methods to provide the input task data (`void setData(Object task)`), to retrieve the result data (`Object getData()`) and to compute result out of task data (`void run()`). This single line of code actually defines the parallel computation to be executed, starts its execution and terminates when the parallel execution is actually terminated. JJPF basic architecture uses two components: clients, that is the user programs, and service, that is distributed server instances that

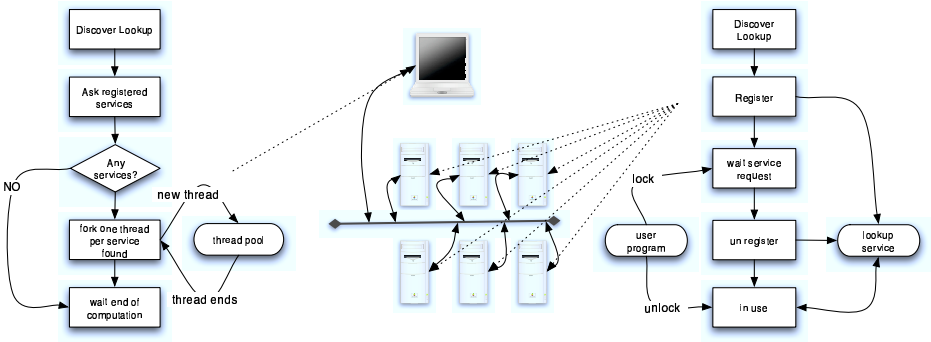

Figure 1. Simplified state diagram for the generic JJPF *client* (left) and *service* (right)

actually compute results out of input task data to execute client programs. Figure 1 sketches the structure of these two components. The client component basically recruits available services and forks a control thread for each one of them. The control thread, in turn, fetches un-computed task items from the task repository, delivers them to the remote service and retrieves the computed results, storing them to the result repository. Service recruiting is performed exploiting JINI. A lookup service is found first, using standard JINI API, then it is queried for available services. Each service descriptor obtained from lookup is passed to a distinct control thread. On the other hand, the service registers to the JINI lookup, and waits for incoming client calls. Once a call is received, it assumes to be recruited by that client, un-registers from the lookup and starts serving task computation requests from the client. This means that clients actually *lock* the services recruited to their exclusive usage. Therefore, in order to use JJPF on a workstation network, the following steps have to be performed: a) JINI has to be installed and configured (this has to be done once and for all, of course), b) JJPF services has to be started at the machines that will eventually be used to run the JJPF distributed server (this also is to be done once and for all), and c) a JJPF client such as the one sketched above has to be prepared, compiled and run on the user workstation. Nothing else is needed. The key concept in JJPF is that service discovery is automatically performed in the client run time support. Not a single line of code dealing with service discovery or recruiting is to be provided by application programmers. Both these mechanisms rely on the JINI technology. This means that all the power of this technology is exploited but also that some limitations of the technology are inherited. In particular, we worried about the fact that JINI discovery mechanisms cannot pass through firewalls, therefore impairing JJPF usability in grid or in large distributed architecture contexts. Indeed, the JINI technology is perfectly suitable to run on workstation clusters within local area networks. JJPF uses two distinct mechanisms to recruit services to clients. It directly requires to the Lookup Service the Service Ids of the available services, i.e. of the nodes currently running the JJPF generic service object, but it also registers to the Lookup Service *observer* objects that will eventually advise the client of new services becoming available, in such a way they can be recruited. When implementing JJPF we had to face the problem of making available user code (the one computing a result out of the single task) to the remote services. JJPF achieves automatic load balancing among the recruited services, due to the scheduling adopted in the control threads managing the remote services. Each control thread fetches tasks to be delivered to the remote nodes from a centralized, synchronized task repository. JJPF also automatically handles faults in service nodes. That is, it takes care of the tasks assigned to a service node in such a way that in case the node does not respond any more they

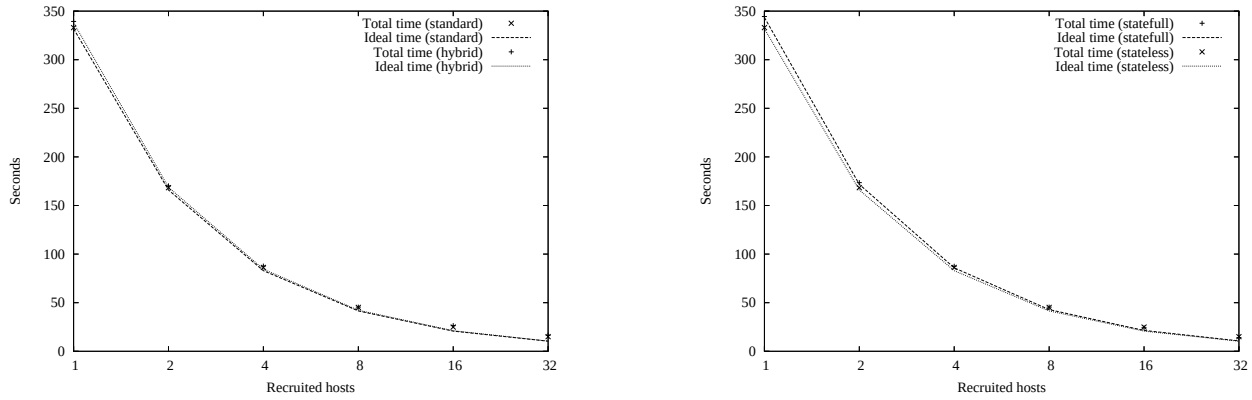

Figure 2. Scalability of JJPF: scalability on a production workstation network, image processing application with standard class loader and with hybrid class loader (left); same application application with (stateful) and without (stateless) access to a shared variable (right)

can be rescheduled to other service nodes, possibly recruited on the fly after realizing the service node fault. This is only possible because of the kind of parallel applications we are taking into account and that are supported in JJPF, that is stream parallel computations. In this case, there are natural *descheduling points* that can be chosen to restart the computation of one of the input tasks, in case of failure of a service node. A trivial one is the start of the computation of the task. Provided that a copy of the task data is kept on the client side (in the control thread, possibly), the task can be rescheduled as soon as the control thread understands that the corresponding service node is death/non responding. This is the choice we actually implemented in JJPF, inheriting the design from `muskel` [12].

3. Experiments

In order to test JJPF features and scalability, we used two kind of applications. Most of the simple scalability measures have actually been performed using a synthetic image processing application. The application just filtered all the images appearing of an image set, applying a sort of blur image filter. This synthetic application mimics real applications that are used, as an example, to pre-process images coming from satellites, telescopes, etc. just before storing them to disks for further, real processing. The image filtering application is actually an embarrassingly parallel application, perfectly matching the task farm pattern. After verifying the scalability and efficiency results of JJPF with the synthetic application, we decided to use a complete application. As there are several people in our department working on web applications, we thought to exploit the available knowledge to develop a page ranking application. The goal was to have a real application at hand that can be used to confirm the scalability and efficiency results achieved with the synthetic case study. The page rank application we developed works with an approximate algorithm. In general, the rank vector x is iteratively computed in such a way that $x^{(k)} = Ax^{k-1}$ until $|| \ x^{(k)} - x^{k-1} \ ||> \epsilon$ [16]. In the approximate algorithm, a pre-processing phase distributes the vector x and the matrix A across a set of services, in such a way that each service can compute a part of the new intermediate rank vector. Once this has been computed, the result is exchanged with the other services in such a way a new iteration (group of iterations) can be computed. The approximate algorithm does not compute the exact page ranking, of course, but the approximation introduced does not impairs the

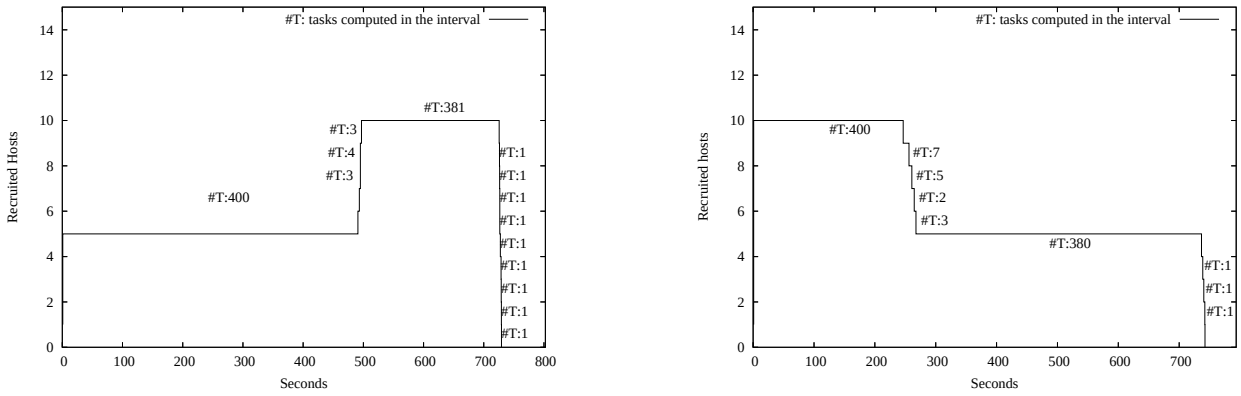

Figure 3. Effect of discovering/recruiting new resources to (left) or dismissing (faulty) resources from (right) the current computation

effectiveness of the algorithm itself. Most page ranking algorithms refer to similar approximation techniques [24,22]. Using these two applications, we run a set of experiments using JJPF to test the feasibility and the efficiency of our approach. We used two distinct kind of distributed architectures: a network of "production" Linux workstations and a cluster of blade PCs, also operated by Linux. The production workstation network was a highly dynamic environment. These workstations are dual boot (Debian Linux and Windows XP), Pentium IV class machines used by the students for their class work. They are often rebooted to switch the operating system and therefore you cannot assume that they stay constantly up and running. Moreover, the users (the students) usually run a variety of tasks on these machines, ranging from WEB browsers to huge compilation and execution tests. The blade machines, on the other side, are based on RLX Pentium III blades, with 3 fast Ethernet networks interconnecting the blades arranged in a single chassis. We have total control on the blade cluster and therefore we could run the tests on "dedicated" nodes. First of all, we tested the scalability of our distributed computation server. We used the synthetic image filtering application to process a stream of input images. The results are shown in Figure 2. In the left part of the Figure, the completion times achieved when the standard class loader mechanism was exploited are shown. In the right part of the Figure, the completion time achieved using our hybrid class loader mechanism is shown. In both cases, we plot the ideal completion time (that is the time spend to compute sequentially all the filtered images divided by the processing elements actually used), the measured completion time and the time actually spent in the computation of the filtered images. Scalability is actually achieved. In all cases, the efficiency was above 90%. Then we measured the efficiency of the recruiting, dismissing mechanisms of JJPF. Therefore we set up two experiments. In the first one, a number of workstations are initially recruited, and further workstations are recruited after that half of the tasks (filtered images) have been already been computed (see Figure 3 left). In the second one, after initially recruiting a number of workstations, half of them are lost (verified faulty) after the computation of half of the tasks (filtered images) (see Figure 3 left). In both cases, the time spent in computing the half tasks with the half workers available took the double of the time taken to compute the other half of the tasks, as expected. The `#T` numbers in the plots, refer to the number of tasks computed in that segment of the plot line. As an example, in the left plot of Figure 3 the `#T:400` indicates that using the initially recruited machines, we computed 400 tasks (filtered images) before actually starting recruiting other machines. When all the additional machines where recruited and just before starting dismissing machines, we computed a further `#T:381` tasks. In

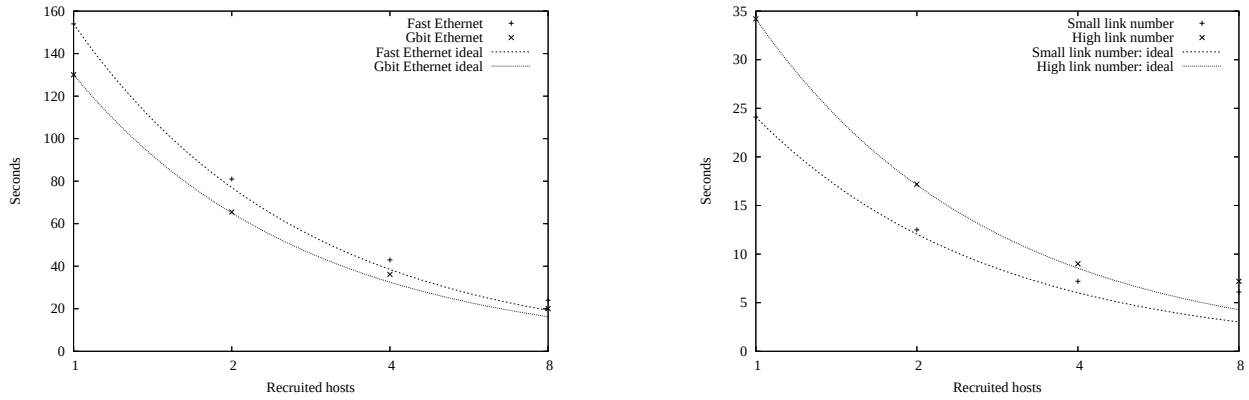

Figure 4. Scalability of JJPF (page rank application): Fast Ethernet vs. Gbit Ethernet (left) Small vs. high link number per page (right)

these experiments, new computing nodes/services are made available for recruitment by running the JJPF run time on new machines and faulty nodes are emulated either by stopping the JJPF support or by unplugging network cables from the switch. Both the experiments whose results are plotted in Figures 2 and 3 have been performed using the network of production workstations. Eventually, we run an experiment with a real application code, the page rank algorithm described above. This experiment has been performed using the blade cluster. We achieved comfortable results, although the scalability measured is not equal to the one achieved using the synthetic, embarrassingly parallel application. The point here is that we need to exchange data among the workers participating in the computation to take care of the approximation algorithm used in the page rank. The right part of the Figure 4, plots the completion times of the page rank application run on a set of 400K pages (therefore a fairly small set of pages) with each page holding a reasonable, but fairly poor number of links to other pages, as well as the completion times achieved using another set with the same number of pages but with pages that hold a quite larger number of links. This was to point out the effect of computation grain on the scalability. In the former case, we compute less before actually starting exchanging the data needed to compute the page rank approximation. In the latter, we compute more. Therefore we pay a smaller (percentage) overhead in the latter case and scalability turns out to be better. In the left part of Figure 4, we point out the differences achieved when using a Gbit Ethernet interconnection between blades instead of a plain Fast Ethernet, 100Mbit interconnection. The network shift improved the completion times, although it did not change significantly the shapes of the completion time curves. Overall, we can state that JJPF demonstrated the expected scalability results as well as its ability too dynamically handle new computational resources, when available, and to safely dismiss nodes (without actually loosing any kind of data), in case they stop working.

4. Related work

Our previous full Java, structured, parallel programming environment `muskel` already provides automatic discovery of computational resource in the context of a distributed workstation network. `muskel` was based on plain RMI Java technology, however and the discovery was simply implemented using multicast datagrams and proper discovery threads. The `muskel` environment also introduces the concept of *application manager* that binds computational resource discovery with autonomic application control in such a way that optimal resource allocation can be dynamically

maintained upon specification by the user of a performance contract to be satisfied [12]. Several other groups proposed or currently propose environments supporting stream parallel computations on workstation networks and clusters. Among the others, we mention Cole's *eSkel* library running on top of MPI [10], Kuchen's C++/MPI skeleton library [20] and CO_2P_2S from the University of Alberta [21]. The former two environments are libraries designed according to the Cole algorithmic skeleton concept. The latter is based on parallel design patterns. None of them allows, at the moment, automatic discovery of computational resources, nor provides fault tolerance features such as those provided by JJPF. The group of Françoise André is currently trying to address the problem of dynamically varying the computational resources assigned to the execution of an SPMD program [6]. This is not actually the same problem we addressed with JJPF, but the techniques used to devise the exact number of resources to be recruited to compute a parallel program are interesting and can be reused in JJPF framework to recruit the right number of service nodes among those available. Our group is also introducing dynamicity handling techniques in the ASSIST environment developed within the GRID.it project [5]. Such techniques are partially derived from the muskel/JJPF experience. The kind of task parallel computations natively supported by JJPF is very close to the one supported by Condor. Condor is a "specialized workload management system for compute-intensive jobs" [11] and "like other full-featured batch systems, it provides a job queuing mechanism, scheduling policy, priority schema, resource monitoring and resource management". However, Condor is a actually a batch system, that is it is not a programming environment, nor it is able to provide (as JJPF does through skeletons and normal form) support for other, different parallelism exploitation patterns/skeletons. Several papers are related to PageRank Algorithm, Haveliwala [18] explores memory-efficient computation, in [19]. Kamvar et al. discuss some methods for accelerating PageRank calculation and in [15] Gleich, Zhukov and Berkhin demonstrate that linear system iterations converge faster than the simple power method and are less sensitive to the changes in teleportation. Rungsawang and Manaskasemsak in [24] e [22] evaluate the performance supplied by an approximated PageRank computation on a Cluster of Workstation using a low-level peer-to-peer MPI implementation.

5. Conclusions

We described JJPF a new distributed server supporting the execution of stream parallel application on workstation networks. The framework exploits plain Java technology, using JINI to address resource discovery. JJPF supports the execution of stream parallel computations using a set of remote service nodes, that is, nodes that basically provide a sort Java interpreter capable of computing generic, user-defined tasks implementing a known interface. Service nodes are discovered and recruited automatically to support user applications. Fault tolerance features have been included in the framework such that the execution of a parallel program can transparently resist to node or network faults. Load balancing is guaranteed across the recruited computational resources, even in case of resources with fairly different computing capabilities. To our knowledge, these features are not present in other distributed parallel programming environments.

References

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of Intl. Conference EuroPar2003: Parallel and Distri buted Computing*, number 2790 in LNCS. Springer, 2003.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in GRID.it. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[3] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimisations. In *Proc. of the IASTED International Conference Parallel and Distributed Computing an d Systems*, pages 955–962. IASTED/ACTA Press, November 1999. Boston, USA.

[4] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[5] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS. Springer Verlag, 2005.

[6] F. André, J. Buisson, and J.L. Pazat. Dynamic adaptation of Parallel Codes: Toward Self-Adaptable Components. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[7] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec. 1999.

[8] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *11th Euromicro Conf. on Parallel, Distributed and Network- Based Processing*, pp 193–200. IEEE, 2003.

[9] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[10] M. Cole and A. Benoit. The eSkel home page, 2005. http://homepages.inf.ed.ac.uk/abenoit1/eSkel/.

[11] Condor community. The CONDOR home page, 2005. http://www.cs.wisc.edu/condor/.

[12] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network -based processing*. IEEE, 2005. Lugano.

[13] I. Foster and C. Kesselman (Editors). *The Grid 2 Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, December 2003.

[14] Ggf community. The Global Grid Forum home page, 2005. http://www.gridforum.org.

[15] D. Gleich, L. Zhukov, and P. Berkhin. Fast Parallel PageRank: A Linear System Approach. Technical report, Yahoo research lab, 2004.

[16] Google community. The Google home page, 2005. http://www.google.com/technology/.

[17] Grid.it community . The GRID.it home page, 2005. http://www.grid.it.

[18] Taher Haveliwala. Efficient Computation of PageRank. TR 1999-31, Stanford Univ., USA, 1999.

[19] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Extrapolation methods for accelerating PageRank computations, Proceedings of the Twelfth Int'l WWW Conference, 2003.

[20] H. Kuchen. A Skeleton Library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. "Springer" Verlag, August 2002.

[21] S. MacDonald, J. Anvik, S. Bromling, J. Scaheffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12), 2002.

[22] Bundit Manaskasemsak and Arnon Rungsawang. Parallel PageRank Computation on a Gigabit PC Cluster. In *AINA (1)*, pages 273–277, 2004.

[23] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[24] Arnon Rungsawang and Bundit Manaskasemsak. PageRank Computation Using PC Cluster. In *PVM/MPI*, pages 152–159, 2003.

[25] Univ. of Tennesse, Mannheim and NERSC/LBNL. The Top500 home page. www.top500.org, 2005.

[26] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 12, December 2002.

[27] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using N etworked Workstations and Parallel Computers*. Prentice Hall, 1999.